

Mobiclip SDK for Nintendo CTR Programming Manual

Version 1.0.7

**The content of this document is highly confidential
and should be handled accordingly.**

Confidential

These coded instructions, statements, and computer programs contain proprietary information of Nintendo and its licensed developers and are protected by national and international copyright laws. They may not be disclosed to third parties or copied or duplicated in any form, in whole or in part, without the prior written consent of Nintendo.

Table of Contents

1	About this document	8
2	Overview	9
2.1	Introduction	9
2.2	Typical Moflex content playback data flow.....	10
2.3	Naming conventions	11
2.4	Modules description	12
2.4.1	mw::mo namespace	12
2.4.2	mw::mo::fastaudio namespace	12
2.4.3	mw::mo::imaadpcm namespace	12
2.4.4	mw::mo::dspadpcm namespace	13
2.4.5	mw::mo::mobiclip namespace.....	13
2.4.6	mw::mo::filter namespace	14
2.4.7	mw::mo::moflex namespace.....	15
2.4.8	mw::mo::helper namespace	16
3	Mobiclip SDK module usage guide	19
3.1	Module initialization and finalization	19
3.2	Module instance creation and release	20
3.3	Module usage derived from mw::mo::GenericVideoDecoder class	21
3.3.1	Video decoder creation	21
3.3.2	Video decoder unpack process	22
3.3.3	Video decoder decoded frame retrieval	23
3.4	Module usage derived from mw::mo::GenericAudioDecoder class	23
3.4.1	Audio decoder creation	24
3.4.2	Audio decoder unpack process.....	24
3.4.3	Audio decoder decoded frame retrieval	25
3.5	Module usage of mw::mo::filter:Deblocker class	25
3.5.1	Deblocker creation	25
3.5.2	Deblocker Apply call	26
3.6	Module usage of mw::mo::moflex:Demuxer class	27
3.6.1	Moflex lexicon.....	27
3.6.2	Demuxer creation	27
3.6.3	Demuxer synchronization process	29
3.6.4	Demuxer streams info retrieval process.....	30
3.6.5	Demuxer read Moflex packet process.....	34
3.6.6	Demuxer stream frame retrieval process	36
3.6.7	Demuxer stream frame removal process	37

3.6.8	Demuxer seeking process	38
4	Mobiclip SDK modules memory usage	40
4.1	mw::mo::mobiclip::Decoder module	40
4.1.1	Methods that perform allocations.....	40
4.1.2	Allocation constraints	40
4.1.3	Number of allocations performed.....	40
4.1.4	Amount of memory allocated	41
4.2	mw::mo::filter::Deblocker module	42
4.2.1	Methods that perform allocations.....	42
4.2.2	Allocation constraints	42
4.2.3	Number of allocations performed.....	42
4.2.4	Amount of memory allocated	43
4.3	mw::mo::imaadpcm::Decoder module.....	44
4.3.1	Methods that perform allocations.....	44
4.3.2	Allocation constraints	44
4.3.3	Number of allocations performed.....	44
4.3.4	Amount of memory allocated	44
4.4	mw::mo::fastaudio::Decoder module.....	45
4.4.1	Methods that perform allocations.....	45
4.4.2	Allocation constraints	45
4.4.3	Number of allocations performed.....	45
4.4.4	Amount of memory allocated	45
4.5	mw::mo::dspadpcm::Decoder module.....	46
4.5.1	Methods that perform allocations.....	46
4.5.2	Allocation constraints	46
4.5.3	Number of allocations performed.....	46
4.5.4	Amount of memory allocated	46
4.6	mw::mo::moflex::Demuxer module	47
4.6.1	Methods that perform allocations.....	47
4.6.2	Allocation constraints	47
4.6.3	Number of allocations performed.....	47
4.6.4	Amount of memory allocated	48
5	Mobiclip player programming and multithreading	49
5.1	Multithreading usage	49
5.2	Multithreaded Mobiclip player structure.....	49
5.2.1	Demux and decode thread.	50
5.2.2	Reader thread.	50
5.2.3	Video presentation thread.....	50
5.2.4	Audio presentation thread.....	50

6	mw::mo::helper namespace API usage guide	51
6.1	Player state graph.....	52
6.1.1	Opening a Moflex content	53
6.1.2	Starting Moflex content playback	54
6.1.3	Stopping Moflex content playback	55
6.1.4	Pausing Moflex content playback.....	55
6.1.5	Resuming Moflex content playback	56
6.1.6	Seeking into Moflex content	57
6.2	Performing video presentation	58
6.2.1	The mw::mo::helper::VideoPresentationBuffer class	58
6.2.2	Warning about video presentation performance	59
6.3	Performing audio presentation.....	59
6.3.1	The mw::mo::helper::AudioPresentationBuffer class	59
6.3.2	Warning about minimum needed audio buffer size	60

Code

Code 2-1 Example of Mobiclip SDK C++ API structure declaration.....	11
Code 2-2 Example of Mobiclip SDK C API structure declaration	11
Code 2-3 Example of Mobiclip SDK C++ API method syntax.....	11
Code 2-4 Example of Mobiclip SDK C API function syntax	11
Code 3-1 Inclusion of Mobiclip.h file in your source code	19
Code 3-2 Example of Mobiclip module initialization.....	19
Code 3-3 Example of Mobiclip module finalization	19
Code 3-4 Example of Mobiclip module creation.....	20
Code 3-5 Example of Mobiclip module release.....	20
Code 3-6 Example of Mobiclip module destructor call	20
Code 3-7 Example of video decoder creation	21
Code 3-8 Example of video decoder unpack process.....	22
Code 3-9 Example of video decoder frame retrieval and removal from the video queue	23
Code 3-10 Example of audio decoder creation.....	24
Code 3-11 Example of audio decoder unpack process.....	24
Code 3-12 Example of audio decoder frame retrieval.....	25
Code 3-13 Example of a deblocker creation	25
Code 3-14 Example of a deblocker Apply call.....	26
Code 3-15 Example of a demuxer creation.....	27
Code 3-16 Example of a synchronization process.....	29
Code 3-17 Example of a stream info retrieval process	30
Code 3-18 Example of a read packet process	34
Code 3-20 Example of a stream frame removal process.....	37
Code 3-211 Example of a low level seeking into a Moflex content	38

Figures

Figure 1 Typical Moflex content playback data flow done by a player	10
Figure 2 Performing a simple playback	52
Figure 3 Create call state graph	53
Figure 4 StartPlayback call state graph	54
Figure 5 StopPlayback call state graph	55
Figure 6 PausePlayback call state graph	55
Figure 7 ResumePlayback call state graph	56
Figure 8 SetPositionByFileOffset call state graph	57

1 About this document

This document is a programming guide that explains basic programming using the Mobiclip SDK for CTR API.

The content of this document is aimed at people who are familiar with C/C++ programming.

For a detailed specification of the Mobiclip SDK API, please see the HTML Function Reference Manual.

If not done yet, we invite you to have a look to the CTR-Mobiclip_SDK_Quickstart_Guide-en_US.pdf document; it will give you a quick overview of Mobiclip SDK for CTR package and its installation steps.

2 Overview

2.1 Introduction

The Mobiclip SDK API is composed of both a C API that conform to the C99 standard and a C++ API.

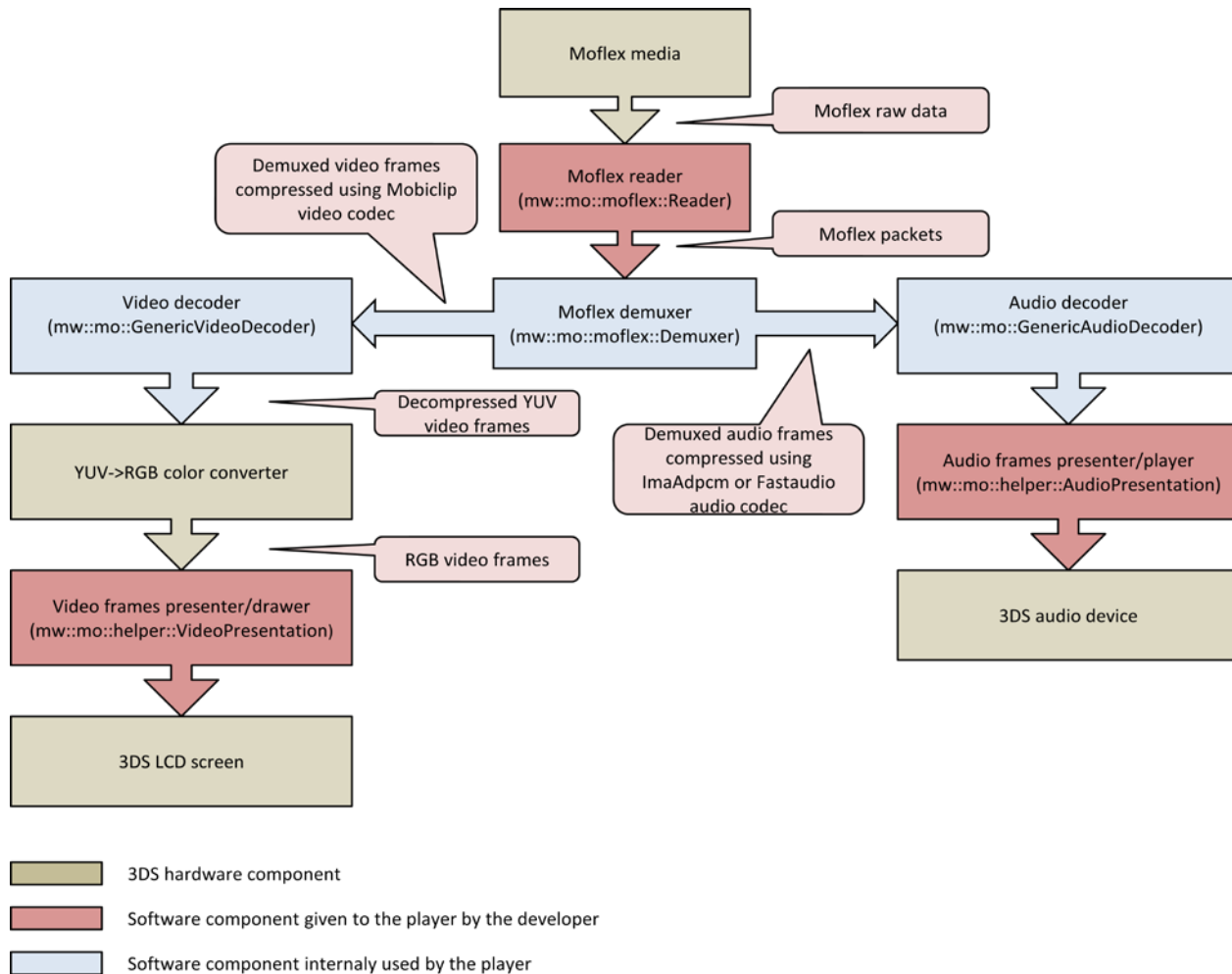
Although the C++ API contains some additional features that can ease the programming work, there is a perfect match between the C++ and the C API functionalities. You'll see in a future chapter the syntactic equivalences between these two APIs.

The Mobiclip SDK API is composed of several modules that can be individually selected depending on developer needs.

2.2 Typical Moflex content playback data flow

Below is a simple schema that shows the data flow and the processes involved in a player when performing a simple Moflex content playback with one video track and one audio track (the most common case that developers will have).

Figure 1 Typical Moflex content playback data flow done by a player



Please have look at Chapter 6 for player implementation details inside the `mw::mo::helper` API.

2.3 Naming conventions

The Mobiclip SDK API is using the same naming convention as the Nintendo CTR SDK API.

The C++ API is using namespaces to categorize objects, functions and types.

The `mw::mo` namespace is the root namespace of Mobiclip SDK C++ API.

On the C API a prefixing is done by concatenating all namespace strings together.

This can easily be shown for a simple structure declaration:

Code 2-1 Example of Mobiclip SDK C++ API structure declaration

```
mw::mo::moflex::Reader reader;
```

Code 2-2 Example of Mobiclip SDK C API structure declaration

```
mwmomoflexReader reader;
```

C functions that use one of the various Mobiclip handles of the API are replaced by C++ object methods without the first parameter of the corresponding C function (that is the Mobiclip handle).

Also output parameters that are retrieved by pointer in the C API are retrieved by reference in the C++ API.

Code 2-3 Example of Mobiclip SDK C++ API method syntax

```
u32 mw::mo::moflex::Demuxer::GetStreamCount(u32& pNbStream);
```

Code 2-4 Example of Mobiclip SDK C API function syntax

```
u32 mwmomoflexDemuxerGetStreamCount(mwmomoflexDemuxerHandle* pHandle, u32* pNbStream);
```

2.4 Modules description

Below you will find a list of the modules that are available in the Mobiclip SDK API, sorted by their corresponding namespace. For most of them there is a corresponding library file that must be included in your `OMakefile` project to correctly build your application.

2.4.1 `mw::mo` namespace

2.4.1.1 `mw::mo::GenericAudioDecoder` class

This is a pure virtual class that exposes a standard interface to an audio decoder. It allows to manage audio decoding with various audio decoder types in a generic way. There is no equivalent feature in the Mobiclip SDK C API.

2.4.1.2 `mw::mo::GenericVideoDecoder` class

This is a pure virtual class that exposes a standard interface to a video decoder. It allows to manage video decoding with various video decoder types in a generic way. There is no equivalent feature in the Mobiclip SDK C API.

2.4.2 `mw::mo::fastaudio` namespace

2.4.2.1 `mw::mo::fastaudio::Decoder` class

This class derives from `mw::mo::GenericAudioDecoder` class and implements its interface for the decoding of FastAudio audio frames. All of its methods have a C equivalent with the `mwmofastaudioDecoder` prefix, like explained in chapter 2.3.

The libraries names to use in your `OMakefile` project are `libmw_mo_FastAudioDecoder.(option).a`.

2.4.3 `mw::mo::imaadpcm` namespace

2.4.3.1 `mw::mo::imaadpcm::Decoder` class

This class derives from `mw::mo::GenericAudioDecoder` class and implements its interface for the decoding of ImaAdpcm audio frames. All of its methods have a C equivalent with the `mwmomaadpcmDecoder` prefix, like explained in chapter 2.3.

The libraries names to use in your `OMakefile` project are `libmw_mo_ImaadpcmDecoder.(option).a`.

2.4.4 mw::mo::dspadpcm namespace

2.4.4.1 mw::mo::dspadpcm::Decoder class

This class derives from `mw::mo::GenericAudioDecoder` class and implements its interface for the decoding of DspAdpcm audio frames. All of its methods have a C equivalent with the `mwmmodspadpcmDecoder` prefix, like explained in chapter 2.3.

The libraries names to use in your `OMakefile` project are `libmw_mo_DspAdpcmDecoder.(option).a`.

2.4.5 mw::mo::mobiclip namespace

2.4.5.1 mw::mo::mobiclip::Decoder class

This class derives from `mw::mo::GenericVideoDecoder` class and implements its interface for the decoding of Mobiclip video frames. All of its methods have a C equivalent with the `mwmomobiclipDecoder` prefix, like explained in chapter 2.3.

The libraries names to use in your `OMakefile` project are `libmw_mo_MobiclipDecoder.(option).a`.

2.4.6 mw::mo::filter namespace

2.4.6.1 mw::mo::filter::YuvImage structure

This structure contains information and data of a YUV420 planar image.

It can be used directly (without copying YUV buffers) with the `nn::y2r` CTR module to perform YUV to RGB color conversion.

It can also be given as input to a `mw::mo::filter::Deblocker` class instance, please see below for more details.

Note: `mw::mo::filter::YuvImage` instances should not be user generated, please use either `mw::mo::GenericVideoDecoder::GetFrame` or `mw::mo::filter::Deblocker::Apply` methods instead.

The Mobiclip video decoder gives YUV data that conform to the ITU-R BT.601 standard.

ITU-R BT.601 defines Y to have a nominal range of 16-235 (black - white), Cb(U) and Cr(V) are defined to have a nominal range of 16 - 240, with 128 corresponding to zero.

The ITU-R BT.601 YUV to RGB conversion formula is:

$$R = 1,164*(Y-16) + 1.596*(Cr-128)$$

$$G = 1,164*(Y-16) - 0.813*(Cr-128) - 0.392*(Cb-128)$$

$$B = 1,164*(Y-16) + 2.017*(Cb-128)$$

2.4.6.2 mw::mo::filter::Deblocker class

This class implements deblocking functionalities of decompressed Mobiclip video frames.

The more compressed a video is, the odds are to see blocking artifacts on screen.

This purpose of this class is to enhance video output quality by removing these blocking artifacts at the expense of additional CPU cost.

All of its methods have a C equivalent with the `mwmofilterDeblocker` prefix, like explained in chapter 2.2.

The libraries names to use in your `OMakefile` project are `libmw_mo_FilterDeblocker.(option).a`.

2.4.7 mw::mo::moflex namespace

2.4.7.1 mw::mo::moflex::Demuxer class

This class implements demuxing features of a Moflex container.

The Moflex container is read and the following information and data can further be extracted:

- information about the type and the number of streams that are stored in the Moflex container.
- the frame data of each stream, for example audio or video frames.

All of its methods have a C equivalent with the `mwmomoflexDemuxer` prefix, like explained in chapter 2.2.

The libraries names to use in your `oMakefile` project are `libmw_mo_MoflexDemuxer.(option).a`.

2.4.8 mw::mo::helper namespace

This namespace contains a collection of classes, structures, enums and constants that are used by all the code samples of the SDK to perform movie playback.

All of its source code is available in the `$(CTRMW_MOBICLIP_SDK_ROOT)/sampledemos/helper` directory.

Part of this collection is shortly described below and can be directly used (or with little modifications) by the developer to perform standard movie playback.

The part that is not described is either meant to be replaced by own developer code (for example audio and video presentation routines) or is used internally by the documented part (for example `mw::mo::helper::VideoConversion` class and `mw::mo::helper::yuvtorgb` namespace).

Although it can greatly ease the developer work the use of these helpers are not mandatory to perform movie playback using this SDK.

The developer is free to use them as is, modify them for its specific needs or not use them at all and directly use the SDK modules documented in the other namespace described previously.

For a detailed specification of the `mw::mo::helper` API, please see the HTML Function Reference Manual.

Also you can have look at Chapter 6 for an introduction to `mw::mo::helper` API usage.

2.4.8.1 mw::mo::helper::PlayerStatus enum

This enum describes the different states the `mw::mo::helper::Player` class can be.

2.4.8.2 mw::mo::helper::PlayerScreen enum

This enum is used to associate a video stream to either top or bottom screen when calling `mw::mo::helper::Player::SetCurrentVideoStream` method.

2.4.8.3 mw::mo::helper::CodecInstantiation class

This class exposes a generic interface for audio and video decoders instantiation. All virtual methods must be implemented by derived classes, like the `mw::mo::helper::MobiclipCodecInstantiation` class.

2.4.8.4 mw::mo::helper::MobiclipCodecInstantiation class

This class implements the interface of the `mw::mo::helper::CodecInstantiation` class. It can recognize/instantiate `mw::mo::imaadpcm::Decoder` and `mw::mo::fastaudio::Decoder` audio decoders and `mw::mo::mobiclip::Decoder` video decoder.

2.4.8.5 mw::mo::helper::VideoPresentationBuffer class

This class exposes a generic interface that manages buffers that will be used for video image presentation to the 3DS screen.

This class cannot be used directly so use derived classes instead.

All virtual methods must be implemented by derived classes to be instantiable.

This class interface is used by the Player class to retrieve information on the target buffers where to make the YUV->RGB color conversion (called screen buffers).

This class interface is also used by the main thread to retrieve the YUV->RGB color converted target buffers (called screen buffers) to display (present) on the 3DS screen.

Please have a look at the mw::mo::helper::GLVideoPresentation class in

`$(CTRMW_MOBICLIP_ROOT)/sampledemos/helpers` directory for an implementation example.

2.4.8.6 mw::mo::helper::AudioPresentationBuffer class

This class exposes a generic interface that manage buffers that will be used for audio presentation (that is playback).

This class cannot be used directly so use derived classes instead.

All virtual methods must be implemented by derived classes to be instantiable.

This class interface is used by the Player class to add new audio data for playback.

Please have a look at the mw::mo::helper::PcmAudioPresentation class in

`$(CTRMW_MOBICLIP_ROOT)/sampledemos/helpers` directory for an implementation example.

2.4.8.7 mw::mo::helper::Player class

This class centralizes all needed mechanisms to perform the playback of a single Moflex content and hides all the complexity of doing that task.

It allows the developer to:

- start/stop playback.
- pause/resume playback.
- perform seeking into the Moflex content.
- retrieve various playback and Moflex content information.

It can currently perform the following tasks:

- play a Moflex content with a 2D video track on top or bottom screen, or in a texture.
- play a Moflex content with a 3D video track on top screen.
- play a Moflex content with a 2D/3D video track on top screen and another 2D video track on bottom screen.
- select and play an audio track from the list of audio tracks available in the Moflex content.

When creating a new instance of the class the developer gives into parameter:

- a mw::mo::moflex::Demuxer instance used for reading, seeking and obtaining information on the Moflex content.
- a mw::mo::helper::VideoPresentationBuffer instance used to draw the video frames on the screen.
- a mw::mo::helper::AudioPresentationBuffer instance used to play the decoded audio frames.
- a mw::mo::helper::CodecInstantiation instance used to instantiate all needed audio and video decoders.

2.4.8.8 mw::mo::helper::FsReader class

This class implements a fully working mw::mo::moflex::Reader structure that can read Moflex content from ROM/MMC/SD medias.

2.4.8.9 mw::mo::helper::RamReader class

This class implements a fully working mw::mo::moflex::Reader structure that can read Moflex content stored in FCRAM memory.

3 Mobiclip SDK module usage guide

To successfully compile an application using Mobiclip SDK API you must include the file `Mobiclip.h` in each of your source code using it, and have its path added to the `INCLUDE` section of your `OMakefile` project.

Code 3-1 Inclusion of `Mobiclip.h` file in your source code

```
#include "Mobiclip.h"
```

Please have a look at the Mobiclip SDK sample codes for additional information on the Mobiclip SDK API usage.

3.1 Module initialization and finalization

Every module of the Mobiclip SDK API described earlier shares a common way to be initialized and finalized. The only exception are the `mw::mo::GenericVideoDecoder` and the `mw::mo::GenericAudioDecoder` pure virtual classes.

Before calling any method of a Mobiclip module you have to call its static `Initialize` method that receives in parameter the allocation/de-allocation functions that will be used by all the module instances.

Code 3-2 Example of Mobiclip module initialization

```
extern void* myMalloc (size_t size); //The allocator function
extern void myFree  (void* ptr);    //The de-allocator function

//Initialization of the mw::mo::mobiclip::Decoder module
mw::mo::mobiclip::Decoder::Initialize(myMalloc, myFree);
```

When all instances of the Mobiclip module are freed the module itself can be finalized by calling the static `Finalize` method of the module.

Code 3-3 Example of Mobiclip module finalization

```
//Finalization of the mw::mo::mobiclip::Decoder module
mw::mo::mobiclip::Decoder::Finalize();
```

3.2 Module instance creation and release

Once a Mobiclip module is initialized you can create instances of it.

Every module of the Mobiclip SDK API has a similar way to create and release module instances.

To use a Mobiclip module instance you have to call its `Create` method. This call will prepare the module instance for its future usage and allocate internal data as needed.

The number of parameters of the `Create` method and their type changes depending on the module type.

The module classes that derive from the `mw::mo::GenericVideoDecoder` class all share the same `Create` method prototype.

In a similar fashion, the module classes that derive from the `mw::mo::GenericAudioDecoder` class all share the same `Create` method prototype.

Code 3-4 Example of Mobiclip module creation

```
mw::mo::mobiclip::Decoder myMobiclipDecoder; //A Mobiclip decoder class instance

//Creation of a mw::mo::mobiclip::Decoder instance for a 400x240 video decoding
myMobiclipDecoder.Create(400, 240, 12);
```

When done with the Mobiclip module instance you release its internal data by calling the `Release` method.

Code 3-5 Example of Mobiclip module release

```
mw::mo::mobiclip::Decoder myMobiclipDecoder; //A Mobiclip decoder class instance

//Release of a mw::mo::mobiclip::Decoder instance
myMobiclipDecoder.Release();
```

If the module instance destructor is called, the `Release` method is automatically called.

Code 3-6 Example of Mobiclip module destructor call

```
mw::mo::mobiclip::Decoder* pMyMobiclipDecoder; //A Mobiclip decoder class instance

//Call of the destructor, the Release method is implicitly called
delete pMyMobiclipDecoder;
```

3.3 Module usage derived from `mw::mo::GenericVideoDecoder` class

There is a common way to use classes derived from `mw::mo::GenericVideoDecoder` class. You can even perform a cast of your video decoder module instance to the `mw::mo::GenericVideoDecoder` type and use it as is: this allows to perform video decoding in generic way, independent of the codec type.

The only video decoder module available in this SDK is Mobiclip, that is the `mw::mo::mobiclip::Decoder` class.

3.3.1 Video decoder creation

The `mw::mo::GenericVideoDecoder::Create` method will allocate internal video buffers necessary for video decoding. Provide the width and height of the video to decode in parameters and the maximum number of decoded video frames that can be pushed.

For the Mobiclip video decoder the constraints on the parameters are:

- video width and height must be a multiple of 16 pixels, the `Create` method will return false otherwise.
- the maximum number of decoded video frames that can be pushed must be at least 6, the `Create` method will return false otherwise. A value of 6 or greater is fine for 2D video . Even if a value of 6 would also work for 3D video, a value of at least 12 is recommended.

Code 3-7 Example of video decoder creation

```
mw::mo::GenericVideoDecoder* pMyVideoDecoder; //A video decoder class instance

// Creation of a mw::mo::GenericVideoDecoder instance for a 400x240 video decoding
// Up to 12 decoded frames can pushed without raising an error
pMyVideoDecoder->Create(400, 240, 12);
```

3.3.2 Video decoder unpack process

The `mw::mo::GenericVideoDecoder::UnpackFrame` method will perform decoding of a compressed frame and push the result in the video queue. If the video queue was full at the time of the call, `false` will be returned and decoding will not be performed.

The compressed frame data pointer given in parameter of the `mw::mo::GenericVideoDecoder::UnpackFrame` method is retrieved by a `mw::mo::moflex::Demuxer::GetStreamFrameData` method call. See later in this document on how to use the `mw::mo::moflex::Demuxer` class.

Code 3-8 Example of video decoder unpack process

```
mw::mo::GenericVideoDecoder* pMyVideoDecoder; //A video decoder class instance
void* pCompressedFrame; // A compressed video frame

int queueCapacity;
pMyVideoDecoder->GetQueueCapacity(queueCapacity);

int pushedFrame;
pMyVideoDecoder->GetNbPushedFrame(pushedFrame);

if (pushedFrame < queueCapacity)
    pMyVideoDecoder->UnpackFrame(pCompressedFrame);
```

3.3.3 Video decoder decoded frame retrieval

The `mw::mo::GenericVideoDecoder::GetFrame` method will retrieve the oldest decoded video frame stored in the video queue, if the video queue is empty the function returns `false`.

The oldest video frame can be later removed from the video queue by calling the `mw::mo::GenericVideoDecoder::PopFrame` method.

Code 3-9 Example of video decoder frame retrieval and removal from the video queue

```
mw::mo::GenericVideoDecoder* pMyVideoDecoder; //A video decoder class instance

u32 pushedFrame;
pMyVideoDecoder->GetNbPushedFrame(pushedFrame);

if (pushedFrame > 0 )
{
    mw::mo::filter::YuvImage yuvImage;
    pMyVideoDecoder->GetFrame(yuvImage);

    // Perform any wanted operation on the retrieved yuv frame before popping it
    pMyVideoDecoder->PopFrame();
}
```

3.4 Module usage derived from `mw::mo::GenericAudioDecoder` class

There is a common way to use classes derived from `mw::mo::GenericAudioDecoder` class. You can even perform a cast of your audio decoder module instance to the `mw::mo::GenericAudioDecoder` type and use it as is: this allows to perform audio decoding in generic way, independent of the codec type.

The audio decoder modules available in this SDK are:

- FastAudio decoder, that is `mw::mo::fastaudio::Decoder` class.
- ImaAdpcm decoder, that is `mw::mo::imaadpcm::Decoder` class.
- DspAdpcm decoder, that is `mw::mo::dspadpcm::Decoder` class.

3.4.1 Audio decoder creation

The `mw::mo::GenericAudioDecoder::Create` method will allocate internal audio buffers necessary for audio decoding. Pass the audio frequency in parameter and the number of channels of the audio stream to decode.

Code 3-10 Example of audio decoder creation

```
mw::mo::GenericAudioDecoder* pMyAudioDecoder; //An audio decoder class instance

// Creation of a mw::mo::GenericAudioDecoder instance for a 22050Hz mono decoding
pMyAudioDecoder->Create(22050, 1);
```

3.4.2 Audio decoder unpack process

The `mw::mo::GenericAudioDecoder::UnpackFrame` method will perform decoding of a compressed frame and keep the decompressed frame in PCM 16 bits format in an internal buffer that is allocated dynamically. If the internal audio buffer can't be allocated, `false` is returned.

There is no queue management like in the `mw::mo::GenericVideoDecoder` class: every `UnpackFrame` call will erase the previously decoded audio content in the internal audio buffer, if any.

The compressed frame data pointer given in parameter of the `mw::mo::GenericVideoDecoder::UnpackFrame` method is retrieved by a `mw::mo::moflex::Demuxer::GetStreamFrameData` method call. See later in this document on how to use the `mw::mo::moflex::Demuxer` class.

Code 3-11 Example of audio decoder unpack process

```
mw::mo::GenericAudioDecoder* pMyAudioDecoder; //An audio decoder class instance
void* pCompressedFrame; // A compressed audio frame
int compressedFrameSize; // The compressed audio frame size

if (pMyAudioDecoder ->UnpackFrame(pCompressedFrame, compressedFrameSize) == false)
{
    // The function raised an error because the internal audio buffer could
    // not be allocated
}
```


3.4.3 Audio decoder decoded frame retrieval

The `mw::mo::GenericAudioDecoder::GetFrame` method will retrieve the pointer of the internal audio buffer and the size of PCM audio sample that it contains, in bytes. You'll have to copy the content if you plan to use it (for audio playback for example), its content will be erased the next time the `mw::mo::GenericAudioDecoder::UnpackFrame` method will be called.

Code 3-12 Example of audio decoder frame retrieval

```
mw::mo::GenericAudioDecoder* pMyAudioDecoder; //An audio decoder class instance

void* pPcmBuffer; // Will contain the pointer of the internal decoded audio buffer
int pcmBufferSize; // Will contain the size of the buffer, in bytes.

// Retrieve pointer and size of the internal decoded audio buffer
pMyAudioDecoder->GetFrame(pPcmBuffer,pcmBufferSize);
```

3.5 Module usage of `mw::mo::filter:Deblocker` class

3.5.1 Deblocker creation

The `mw::mo::filter::Deblocker::Create` method will allocate internal buffers necessary for the deblocking process. There is no parameter to give to this function.

Code 3-13 Example of a deblocker creation

```
mw::mo::filter::Deblocker myDeblocker; //A deblocker instance

// Creation of a mw::mo::filter::Deblocker instance
myDeblocker.Create();
```

3.5.2 Deblocker Apply call

The `mw::mo::filter::Deblocker::Apply` method takes a `mw::mo::filter::YuvImage` as input parameter and gives a deblocked `mw::mo::filter::YuvImage` as output parameter. Additional data can be allocated internally that are necessary for the deblocking process, the error `mw::mo::filter::DEBLOCKER_RETURN_NOT_ENOUGH_MEMORY` will be returned if not enough memory is found.

The input and output `mw::mo::filter::YuvImage` parameters must be different when using a `mw::mo::filter::YuvImage` structure retrieved from a Mobiclip video decoder as input or the `mw::mo::filter::DEBLOCKER_RETURN_NOT_WRITABLE` error will be returned.

Code 3-14 Example of a deblocker Apply call

```
mw::mo::filter::Deblocker myDeblocker; //A deblocker instance
mw::mo::GenericVideoDecoder* pMyVideoDecoder; //A video decoder class instance

u32 pushedFrame;
pMyVideoDecoder->GetNbPushedFrame(pushedFrame);
if (pushedFrame > 0 )
{
    mw::mo::filter::YuvImage yuvImage;
    pMyVideoDecoder->GetFrame(yuvImage);

    mw::mo::filter::YuvImage deblockedYuvImage;
    mw::mo::filter::DeblockerReturn ret;

    ret = myDeblocker.Apply(yuvImage, deblockedYuvImage);
    if (ret != mw::mo::filter::DEBLOCKER_RETURN_OK)
    {
        // Apply returned an error
    }
}
```

3.6 Module usage of mw::mo::moflex:Demuxer class

3.6.1 Moflex lexicon

3.6.1.1 Moflex container

A Moflex container is a stream of bytes that conforms to the Moflex format.

Most often a Moflex container is stored in the form of a file (we usually give it a .moflex extension string) that is stored in media like CTR ROM or from MMC/SD/SDHC cards.

3.6.1.2 Moflex packet

The Moflex container is a sequence of Moflex packets. A Moflex packet can contain several Moflex contents of various types, while a Moflex content can also be stored across several Moflex packets.

3.6.1.3 Moflex content

There are two main different Moflex contents that are exposed by the Demuxer API:

- **Moflex synchro block:** it contains useful information about the number and the type of Moflex streams that are stored in the Moflex container.
- **Moflex stream:** it can be of type audio, video or timeline for example. A Moflex stream can contain one or more Moflex frames.

3.6.1.4 Moflex frame

A Moflex frame contains data that can be independently decoded and/or used. For example a video frame holds enough data to decode a full video image.

3.6.2 Demuxer creation

The `mw::mo::moflex::Demuxer::Create` method will allocate internal buffers necessary for the demuxing process.

The method parameters are:

- a `mw::mo::moflex::Reader` structure that contains a collection of functions that manages reading from a specific media, like from CTR ROM or from MMC/SD/SDHC cards. Please have a look at the helper `mw::mo::helper::FsReader` class source code for an implementation example of a `mw::mo::moflex::Reader` structure.
- a cookie, this is a void pointer that contains data associated to a Moflex container and that will be used by the `mw::mo::moflex::Reader` structure.

Code 3-15 Example of a demuxer creation

```
#include "helper_FsReader.h" // This is to use the FsReader helper
mw::mo::helper::FsReader fsReader; // A FsReader instance
```

```
mw::mo::moflex::Demuxer myDemuxer; //A demuxer instance

// Creation of a mw::mo::filter::Deblocker instance
myDemuxer.Create(fsReader.GetMoflexReader(), fsReader.GetCookie());
```

3.6.3 Demuxer synchronization process

When a demuxer object is newly created with the `mw::mo::moflex::Demuxer::Create` method or each time seeking into the Moflex container is performed by using the

`mw::mo::moflex::Demuxer::SetContentPosition` method, finding a synchro block in the Moflex container is the first thing that has to be done.

Once a synchro block is found, some useful information on the number and the type of streams that are stored in the Moflex container can be retrieved.

Code 3-16 Example of a synchronization process

```
mw::mo::moflex::Demuxer myDemuxer; //A demuxer instance
mw::mo::moflex::Demuxer demuxerRet; //A demuxer return value

do
{
    demuxerRet = myDemuxer.ReadPacket();
}
while (demuxerRet == mw::mo::moflex::DEMUXER_RETURN_SYNCHRONISING ||
        demuxerRet == mw::mo::moflex::DEMUXER_RETURN_BUSY);

switch(demuxerRet)
{
    case mw::mo::moflex::DEMUXER_RETURN_OK:
        // Synchronization done
        break;
    case mw::mo::moflex::DEMUXER_RETURN_END:
        // Synchronization not done, end of stream reached
        break;
    default:
        // An error was raised
        break;
}
```

3.6.4 Demuxer streams info retrieval process

3.6.4.1 General mw::mo::moflex::StreamInfo retrieval process

When synchronization is done, information about the number of streams, their type and characteristics can be retrieved by using `mw::mo::moflex::Demuxer::GetStreamCount` and `mw::mo::moflex::Demuxer::GetStreamInfo` methods.

Code 3-17 Example of a stream info retrieval process

```
mw::mo::moflex::Demuxer myDemuxer; //A demuxer instance

u32 nbStreams;
myDemuxer.GetStreamCount(nbStreams);
for (u32 i=0; i<nbStreams; i++)
{
    mw::mo::moflex::StreamInfo stream;
    myDemuxer.GetStreamInfo(i, stream);
    switch(stream.m_Generic.m_StreamType)
    {
        case mw::mo::moflex::STREAM_TYPE_VIDEO:
            // The stream is of type video
            break;
        case mw::mo::moflex::STREAM_TYPE_AUDIO:
            // The stream is of type audio
            break;
        case mw::mo::moflex::STREAM_TYPE_TIMELINE:
            // The stream is of type timeline
            break;
    }
}
```

3.6.4.2 mw::mo::moflex::StreamInfo description

`mw::mo::moflex::StreamInfo` is an union of the following structures:

- `mw::mo::moflex::GenericStreamInfo`
- `mw::mo::moflex::VideoStreamInfo`
- `mw::mo::moflex::AudioStreamInfo`
- `mw::mo::moflex::TimelineStreamInfo`

These will be described below.

3.6.4.3 mw::mo::moflex::GenericStreamInfo description

mw::mo::moflex::GenericStreamInfo contains fields that are shared by all mw::mo::moflex::StreamInfo structure types. It contains the following fields:

- m_StreamType: this field contains the type of the mw::mo::moflex::StreamInfo structure. It can be mw::mo::moflex::STREAM_TYPE_VIDEO, mw::mo::moflex::STREAM_TYPE_AUDIO or mw::mo::moflex::STREAM_TYPE_TIMELINE.

3.6.4.4 mw::mo::moflex::VideoStreamInfo description

mw::mo::moflex::VideoStreamInfo structure contains general information on a video stream:

- m_StreamType: its value must be mw::mo::moflex::STREAM_TYPE_VIDEO.
- m_VideoCodec: the video codec used, its value should be mw::mo::moflex::VIDEO_CODEC_MOBICLIP.
- m_Layout: the stereoscopic layout of the video stream, if any. It can have the following values:
 - mw::mo::moflex::LAYOUT_NONE: video stream is 2D.
 - mw::mo::moflex::LAYOUT_INTERLEAVED_LEFT_EYE_FIRST: video stream is stereoscopic where each eye is encoded in separated frames that are interleaved in the Moflex container, the first encoded video frame being the left eye.
 - mw::mo::moflex::LAYOUT_INTERLEAVED_RIGHT_EYE_FIRST: video stream is stereoscopic where each eye is encoded in separated frames that are interleaved in the Moflex container, the first encoded video frame being the right eye.
 - mw::mo::moflex::LAYOUT_TOPTOBOTTOM_LEFT_EYE_FIRST: video stream is stereoscopic where both eyes are encoded in the same frame on top of each other, the top part being the left eye.
 - mw::mo::moflex::LAYOUT_TOPTOBOTTOM_RIGHT_EYE_FIRST: video stream is stereoscopic where both eyes are encoded in the same frame on top of each other, the top part being the right eye.
 - mw::mo::moflex::LAYOUT_SIDE_BY_SIDE_LEFT_EYE_FIRST: video stream is stereoscopic where both eyes are encoded in the same frame side by side, the left part being the left eye.
 - mw::mo::moflex::LAYOUT_SIDE_BY_SIDE_RIGHT_EYE_FIRST: video stream is stereoscopic where both eyes are encoded in the same frame side by side, the left part being the right eye.

- `m_Rotation`: the pre-rotation done on the video stream. It can have the following values:
 - `mw::mo::moflex::ROTATION_NONE`: video stream is not pre-rotated.
 - `mw::mo::moflex::ROTATION_CLOCKWISE_90`: video stream is pre-rotated by 90° clockwise. This kind of pre-rotation is used to allow rendering of the video stream directly on the display frame buffers (display buffers are 400x240 pixels rotated clockwise, that is 240x400).
 - `mw::mo::moflex::ROTATION_CLOCKWISE_180`: video stream is pre-rotated by 180° clockwise. It has been added for potential future applications but has no practical use for the moment.
 - `mw::mo::moflex::ROTATION_CLOCKWISE_270`: video stream is pre-rotated by 270° clockwise. It has been added for potential future applications but has no practical use for the moment.
- `m_Width`: the width of the video stream in pixels.
- `m_Height`: the height of the video stream in pixels.
- `m_FpsRate`: the fps rate of the video stream.
- `m_FpsScale`: the fps scale of the video stream, the actual fps of the video stream is retrieved by the following computation: `m_FpsRate/m_FpsScale`.
- `m_PelRatioRate`: the video stream pixel aspect ratio rate. Most of the time it has a value of 1.
- `m_PelRatioScale`: the video stream pixel aspect ratio scale. Most of the time it has a value of 1.
The actual pixel aspect ratio of the video stream is retrieved by the following computation: `m_PelRatioRate/m_PelRatioScale`, if resulting value is different from 1 then the video stream is anamorphosed. A value of 1 should be expected for the moment.

3.6.4.5 mw::mo::moflex::AudioStreamInfo description

mw::mo::moflex::AudioStreamInfo structure contains general information on an audio stream:

- m_StreamType: its value must be mw::mo::moflex::STREAM_TYPE_AUDIO.
- m_AudioCodec: the audio codec used, its value can be:
 - mw::mo::moflex::AUDIO_CODEC_FASTAUDIO: audio stream is encoded in FastAudio audio format.
 - mw::mo::moflex::AUDIO_CODEC_IMAADPCM: audio stream is encoded in ImaAdpcm audio format.
 - mw::mo::moflex::AUDIO_CODEC_DSPADPCM: audio stream is encoded in DspAdpcm audio format.
 - mw::mo::moflex::AUDIO_CODEC_PCM: audio stream is encoded in PCM 16 bits audio format.
- m_Frequency: the audio stream frequency.
- m_Channel: the number of channels of the audio stream.

3.6.4.6 mw::mo::moflex::TimelineStreamInfo description

mw::mo::moflex::AudioStreamInfo structure contains general information on a timeline stream:

- m_StreamType: its value must be mw::mo::moflex::STREAM_TYPE_TIMELINE.
- m_AssociatedStreamIndex: the video stream index that the timeline stream is managing.

3.6.5 Demuxer read Moflex packet process

The process of reading a Moflex container is to repeatedly call the `mw::mo::moflex::Demuxer::ReadPacket` method. Each time the `mw::mo::moflex::Demuxer::ReadPacket` method returns `mw::mo::moflex::DEMUXER_RETURN_OK`, a full Moflex packet is read and the following internal process is done:

- Moflex packet are stored in an internal queue, each Moflex packet containing a fragment of one or several streams frame data,.
- when there are enough Moflex packets to reconstruct at least a full stream frame (which can be of type audio, video or timeline for example) they are reconstructed and stored in the demuxer object. The Moflex packets that were used for the frame reconstruction are removed from the demuxer internal queue.
- these frames will remain in the demuxer object and use main memory resources until explicitly removed by calling the `mw::mo::moflex::Demuxer::PopStreamFrame` method.

Code 3-18 Example of a read packet process

```
mw::mo::moflex::Demuxer myDemuxer; //A demuxer instance
mw::mo::moflex::Demuxer demuxerRet; //A demuxer return value

do
{
    demuxerRet = myDemuxer.ReadPacket();
    switch(demuxerRet)
    {
        case mw::mo::moflex::DEMUXER_RETURN_OK:
            // A Moflex packet is read, new frames may be available
            break;
        case mw::mo::moflex::DEMUXER_RETURN_END:
            // End of Moflex content reached
            break;
        case mw::mo::moflex::DEMUXER_RETURN_SYNCHRONISING:
            // Still searching for a Moflex synchro block
            break;
        case mw::mo::moflex::DEMUXER_RETURN_BUSY:
            // Busy reading a Moflex packet
            break;
        default:
            // An error was raised
            break;
    }
}
```

Note: Care should be taken when calling `mw::mo::moflex::Demuxer::ReadPacket` method repeatedly. Indeed, the number of enqueued frames in the demuxer object can quickly grow beyond reasonable, dramatically increasing memory usage.

The proper way of doing it is to check the number of enqueued frames in the demuxer object with `mw::mo::moflex::Demuxer::GetStreamFrameCount` method and then decide on whether to call `mw::mo::moflex::Demuxer::ReadPacket` method based on the returned value.

In the case of reading a Moflex file from a physical media like CTR ROM (the most common scenario) `mw::mo::moflex::Demuxer::ReadPacket` method can be called only if there are no enqueued frames in the demuxer object; this is what is done in the `mw::mo::helper::Player` class.

In future revisions of the SDK we may add an option in the `mw::mo::moflex::Demuxer` class that would limit the maximum number frames that could be enqueued in a demuxer object; this would let the developer call freely the `mw::mo::moflex::Demuxer::ReadPacket` method without having to manually check the number of enqueued frames. Please note however that this may change the behavior of existing developer player code in some rare cases.

3.6.6 Demuxer stream frame retrieval process

The demuxer object maintains queues of fully read stream frames. There are as many queues as streams in the Moflex container.

A stream frame can be retrieved by using the `mw::mo::moflex::Demuxer::GetStreamFrameData` method, giving in parameter:

- an `int` containing the requested stream index from where to retrieve the frame.
- a reference to an output `u8` pointer what will contain the frame data.
- a reference to an output `u32` integer that will contain the frame data size in bytes.
- a reference to an output `u32` integer that will contain the frame type, which can either be `mw::mo::moflex::FRAME_TYPE_IFRAME` or `mw::mo::moflex::FRAME_TYPE_PFRAME`.
- a reference to an output `s64` integer that will contain the frame timestamp in microseconds.

Code 3-19 Example of a stream frame retrieval process

```
mw::mo::moflex::Demuxer myDemuxer; //A demuxer instance
u32 streamIndex; // The wanted stream index

u32 nbFrames;
myDemuxer.GetStreamFrameCount(streamIndex, nbFrames);
if (nbFrames > 0) // There is at least 1 frame of the stream 'streamIndex'
{
    const u8* pFrameData;
    u32 frameSize;
    u32 frameType;
    s64 frameTimeStamp;

    myDemuxer.GetStreamFrameData(streamIndex, pFrameData, frameSize,
                                frameType, frameTimeStamp);

    // Do your frame data processing here
}
```

3.6.7 Demuxer stream frame removal process

When no longer needed, a stream frame can be removed from its corresponding demuxer internal queue by using the `mw::mo::moflex::Demuxer::PopStreamFrame` method, giving in parameter:

- an int containing the stream index from where to remove the frame.

Code 3-20 Example of a stream frame removal process

```
mw::mo::moflex::Demuxer myDemuxer; //A demuxer instance
u32 streamIndex; // The wanted stream index

u32 nbFrames;
myDemuxer.GetStreamFrameCount(streamIndex, nbFrames);
if (nbFrames > 0) // There is at least 1 frame of the stream 'streamIndex'
{
    myDemuxer.PopStreamFrame(streamIndex);
}
```

Note: Keeping many frames (like hundreds of frames) in the demuxer queues is not advised: this will dramatically increase memory usage of the demuxer and also have adverse effects in term of performance when calling the `mw::mo::moflex::Demuxer::Release` method. The time spent on this method is proportional to the maximum number of frames that have been kept enqueued at any given time; as example this method can take hundreds of microseconds to execute if hundreds of frames are kept enqueued in the demuxer module.

3.6.8 Demuxer seeking process

3.6.8.1 Low level seeking into a Moflex container using a raw offset

This method allows to seek at a specified position into the Moflex container. A low level seeking process is performed by using the `mw::mo::moflex::Demuxer::SetContentPosition` method, giving in parameter:

- an `s64` containing the raw offset value, that is the position in bytes from the beginning of the container where we want to seek.

The demuxer will seek to this position then parse the Moflex container until it finds a position where it can restart the demux process, that is when it finds a Moflex synchro bloc (this process is done in the `mw::mo::moflex::Demuxer::ReadPacket` method) .

Code 3-211 Example of a low level seeking into a Moflex content

```
mw::mo::moflex::Demuxer myDemuxer; //A demuxer instance
s64 seekOffset; // The wanted position where we want to seek

myDemuxer.SetContentPosition (seekOffset);
```

3.6.8.2 High level seeking into a Moflex container using timeline information

This method allows to jump to a specified video keyframe and as such is more suitable to perform fast and precise seeking: it gives valuable information on where to seek in Moflex container (still by using the `mw::mo::moflex::Demuxer::SetContentPosition` method described in the previous chapter).

To do that the Moflex container must have a timeline stream embedded in it, its stream type is `mw::mo::moflex::STREAM_TYPE_TIMELINE`. The timeline stream is always the first stream written inside the Moflex container, as a result it is also the first stream that is available when demuxing a Moflex content.

A timeline stream is composed of two parts:

- a timeline header structure of type `mw::mo::moflex::TimelineHeader`.
- an array of timeline entries structures of type `mw::mo::moflex::TimelineEntry`.

The timeline header contains the following information:

- an u32 containing the number of timeline entries (one for each video keyframe) that will follow this header.
- an u32 containing the total number of video frames.
- an s64 containing the video stream duration in microseconds.

A timeline entry contains the following information:

- an u32 containing the frame number of the associated keyframe video frame.
- an s64 containing the timestamp in microseconds of the associated keyframe video frame.
- an s64 containing the position in bytes from the beginning of the container where the associated keyframe video frame is.

By using timeline information you can easily seek to a specified keyframe using various schemes:

- by using an index inside the timeline entry array.
- by parsing the timeline entry array and searching for a keyframe that is close to a given frame number.
- by parsing the timeline entry array and searching for a keyframe that is close to a given timestamp.

4 Mobiclip SDK modules memory usage

This chapter describes the memory usage of all Mobiclip SDK modules.

Please note that the amount of memory used by each module described in this chapter can be higher by a small amount in practice; this can be due to heap management and alignment constraints when allocating data.

Also the memory configuration of the code samples (that is the size of the heaps that are used) provided with this SDK are tuned to allow video playback of the major combinations of video resolutions and/or layout. Depending on the movies used in the application the amount of memory (that is the size of the heaps) used by Mobiclip modules can be lowered or have to be raised.

4.1 mw::mo::mobiclip::Decoder module

4.1.1 Methods that perform allocations

Allocations are performed only in the `mw::mo::mobiclip::Decoder::Create` method.

4.1.2 Allocation constraints

The allocators functions given in parameter to the `mw::mo::mobiclip::Decoder::Initialize` method must conform to the following constraints:

- allocated data must reside in CTR device memory.
- allocated data must be at least 4 bytes aligned.

4.1.3 Number of allocations performed

Given the width, height and `maxBufferedFrames` parameters given to the `mw::mo::mobiclip::Decoder::Create` method, the number of allocations performed by the `mw::mo::mobiclip::Decoder` module is:

`maxBufferedFrames + 5`

4.1.4 Amount of memory allocated

Given the width, height and `maxBufferedFrames` parameters given to the `mw::mo::mobiclip::Decoder::Create` method, the amount of memory allocated by the `mw::mo::mobiclip::Decoder` module (when using release version of the library) is:

$$1708 + \text{maxBufferedFrames} * 12 + \text{maxBufferedFrames} * (\text{stride} * (\text{height} + \text{height}/2 + 1))$$

When using debug or development version libraries 28 additional bytes are allocated to store various statistic values.

The value of `stride` is computed as is:

```
if width <= 256
    stride = 256
else if width <= 512
    stride = 512
else
    stride = 1024
```

4.1.4.1 Examples

For 400x240 videos and `maxBufferedFrames == 12` the memory usage is:

$$1708 + 12 * 12 + 12 * (512 * (240 + 240/2 + 1)) \approx 2168 \text{ KiB}$$

For 320x240 videos and `maxBufferedFrames == 12` the memory usage is:

$$1708 + 12 * 12 + 12 * (512 * (240 + 240/2 + 1)) \approx 2168 \text{ KiB}$$

For 240x400 videos (400x240 pre-rotated by 90° clockwise) and `maxBufferedFrames == 12` the memory usage is:

$$1708 + 12 * 12 + 12 * (256 * (400 + 400/2 + 1)) \approx 1805 \text{ KiB}$$

For 240x320 videos (320x240 pre-rotated by 90° clockwise) and `maxBufferedFrames == 12` the memory usage is:

$$1708 + 12 * 12 + 12 * (256 * (320 + 320/2 + 1)) \approx 1805 \text{ KiB}$$

Note: As the numbers show, the amount of memory used is significantly lower when using pre-rotated videos; using pre-rotated videos is a good and simple way to significantly lower your movie playback memory footprint.

4.2 mw::mo::filter::Deblocker module

4.2.1 Methods that perform allocations

Allocations are performed in the `mw::mo::filter::Deblocker::Create` and `mw::mo::mobiclip::Decoder::Apply` methods.

4.2.2 Allocation constraints

The allocators functions given in parameter of the `mw::mo::filter::Deblocker::Initialize` method must conform to the following constraints:

- allocated data must reside in CTR device memory.
- allocated data must be at least 4 bytes aligned.

4.2.3 Number of allocations performed

Unless the field values of the first `mw::mo::filter::YuvImage` structure given to the `mw::mo::filter::Deblocker::Apply` method change, the number of allocations performed by the `mw::mo::filter::Deblocker` module is:

- 1 in the `mw::mo::filter::Deblocker::Create` method
- 1 allocation in the `mw::mo::filter::Deblocker::Apply` method if its `deblockerQuality` parameter value never changes.
- 2 allocations in the `mw::mo::filter::Deblocker::Apply` method if its `deblockerQuality` parameter value changes from `mw::mo::filter::DEBLOCKER_QUALITY_MEDIUM` value to `mw::mo::filter::DEBLOCKER_QUALITY_HIGH` value.

In general an allocation will be performed in `mw::mo::filter::Deblocker::Apply` method each time its internal buffer is too small to fit the target deblocked image. This can happen if you suddenly call `mw::mo::filter::Deblocker::Apply` method with a bigger `mw::mo::filter::YuvImage` as input.

4.2.4 Amount of memory allocated

From the `m_Stride` and `m_Height` fields of the first `mw::mo::filter::YuvImage` structure given to the `mw::mo::filter::Deblocker::Apply` method, the amount of memory allocated by the `mw::mo::filter::Deblocker` module for the different values of the `deblockerQuality` parameter values of the `mw::mo::filter::Deblocker::Apply` method is (when using release version of the library):

If `deblockerQuality` parameter value is `mw::mo::filter::DEBLOCKER_QUALITY_HIGH`:

$$8 + (m_Stride * (m_Height + m_Height/2 + 1))$$

If `deblockerQuality` parameter value is `mw::mo::filter::DEBLOCKER_QUALITY_MEDIUM`:

$$8 + (m_Stride * (m_Height + 1))$$

When using debug or development version libraries 32 additional bytes are allocated to store various statistic values.

4.2.4.1 Examples

For 400x240 videos the memory usage is:

$$8 + (512 * (240 + 240/2 + 1)) \approx 181 \text{ KiB}$$

For 320x240 videos the memory usage is:

$$8 + (512 * (240 + 240/2 + 1)) \approx 181 \text{ KiB}$$

For 240x400 videos (400x240 pre-rotated by 90° clockwise) the memory usage is:

$$8 + (256 * (400 + 400/2 + 1)) = 153864 \approx 151 \text{ KiB}$$

For 240x320 videos (320x240 pre-rotated by 90° clockwise) the memory usage is:

$$8 + (256 * (320 + 320/2 + 1)) = 123144 \approx 121 \text{ KiB}$$

Note: As these numbers show, the amount of memory used is significantly lower when using pre-rotated videos; using pre-rotated videos is a good and simple way to significantly lower your movie playback memory footprint.

4.3 mw::mo::imaadpcm::Decoder module

4.3.1 Methods that perform allocations

Allocations are performed in the `mw::mo::imaadpcm::Decoder::Create` and `mw::mo::imaadpcm::Decoder::Unpack` methods.

4.3.2 Allocation constraints

The allocators functions given in parameter of the `mw::mo::imaadpcm::Decoder::Initialize` method must conform to the following constraints:

- allocated data can reside in CTR system memory.
- allocated data must be at least 4 bytes aligned.

4.3.3 Number of allocations performed

The `mw::mo::imaadpcm::Decoder::Create` method does 1 allocation.

At least 1 allocation is performed in the `mw::mo::imaadpcm::Decoder::Unpack` method (most of the time 2 allocations will be performed during Moflex movie playback).

4.3.4 Amount of memory allocated

From the `frequency` and `nbChannel` parameters given to the `mw::mo::imaadpcm::Decoder::Create` method and `videoFps` (a floating point value that stores the frame rate of the associated video stream, the amount of memory allocated by the `mw::mo::imaadpcm::Decoder` module is (when using release version of the library):

$$40 + \text{ceil}f(((\text{float})\text{frequency}/\text{videoFps})/256.f) * 512 * \text{nbChannel}$$

When using debug or development version libraries 32 additional bytes are allocated to store various statistic values.

4.3.4.1 Examples

For 30fps videos and an audio at 22050Hz stereo, the maximum memory usage is:

$$40 + \text{ceil}f((22050.f / 30.f)/256.f) * 512 * 2 \approx 3 \text{ KiB}$$

4.4 mw::mo::fastaudio::Decoder module

4.4.1 Methods that perform allocations

Allocations are performed in the `mw::mo::fastaudio::Decoder::Create` and `mw::mo::fastaudio::Decoder::Unpack` methods.

4.4.2 Allocation constraints

The allocators functions given in parameter of the `mw::mo::fastaudio::Decoder::Initialize` method must conform to the following constraints:

- allocated data can reside in CTR system memory.
- allocated data must be at least 4 bytes aligned.

4.4.3 Number of allocations performed

The `mw::mo::fastaudio::Decoder::Create` method does 2 allocations.

At least 1 allocation is performed in the `mw::mo::fastaudio::Decoder::Unpack` method (most of the time 2 allocations will be performed during Moflex movie playback).

4.4.4 Amount of memory allocated

From the `frequency` and `nbChannel` parameters given to the `mw::mo::fastaudio::Decoder::Create` method and `videoFps` (a floating point value that stores the frame rate of the associated video stream, the amount of memory allocated by the `mw::mo::fastaudio::Decoder` module is (when using release version of the library):

$$24 + 464 * nbChannel + \text{ceil}f(((\text{float})\text{frequency}/\text{videoFps})/256.f) * 512 * nbChannel$$

When using debug or development version libraries 32 additional bytes are allocated to store various statistic values.

4.4.4.1 Examples

For 30fps videos and an audio at 22050Hz stereo, the maximum memory usage is:

$$24 + 464 * 2 + \text{ceil}f((22050.f / 30.f)/256.f) * 512 * 2 \approx 3.9 \text{ KiB}$$

4.5 mw::mo::dspadpcm::Decoder module

4.5.1 Methods that perform allocations

Allocations are performed in the `mw::mo::dspadpcm::Decoder::Create` and `mw::mo::dspadpcm::Decoder::Unpack` methods.

4.5.2 Allocation constraints

The allocators functions given in parameter of the `mw::mo::dspadpcm::Decoder::Initialize` method must conform to the following constraints:

- allocated data can reside in CTR system memory.
- allocated data must be at least 4 bytes aligned.

4.5.3 Number of allocations performed

The `mw::mo::dspadpcm::Decoder::Create` method does 1 allocation.

The `mw::mo::dspadpcm::Decoder::Unpack` method performs 1 allocation for mono audio and 2 allocations for multi-channel audio (regardless of the actual channel count).

4.5.4 Amount of memory allocated

From the `frequency` and `nbChannel` parameters given to the `mw::mo::dspadpcm::Decoder::Create` method and `videoFps` (a floating point value that stores the frame rate of the associated video stream), the amount of memory allocated by the `mw::mo::dspadpcm::Decoder` module can be computed as follows:

$$\text{maxSamples} = \text{ceil}(\text{frequency} / \text{videoFps} * 1024.f) * 1024$$

In mono the amount of memory allocated is:

$$\text{memUsage} = 28 + (\text{maxSamples} + 32) * 2$$

In multi-channel, an additional amount is allocated:

$$\text{memUsage} += \text{maxSamples} * 2 * \text{nbChannel}$$

When using debug or development version libraries 28 additional bytes are allocated to store various statistic values.

4.5.4.1 Examples

For a 30fps video and an audio at 22050Hz mono, the maximum memory usage is:

$$\text{maxSamples} = \text{ceilf}(22050.f / 30.f * 1024.f) * 1024 = 1024$$
$$\text{memUsage} = 28 + (1024 + 32) * 2 \approx 2 \text{ KiB}$$

For a 25fps video and an audio at 44100Hz mono, the maximum memory usage is:

$$\text{maxSamples} = \text{ceilf}(44100.f / 25.f * 2048.f) * 1024 = 2048$$
$$\text{memUsage} = 28 + 2048 * 2 * 2 + (2048 + 32) * 2 \approx 12 \text{ KiB}$$

4.6 mw::mo::moflex::Demuxer module

4.6.1 Methods that perform allocations

Allocations are performed in the `mw::mo::moflex::Demuxer::Create` and

`mw::mo::moflex::Demuxer::ReadPacket` methods.

4.6.2 Allocation constraints

The allocators functions given in parameter of the `mw::mo::moflex::Demuxer::Initialize` method must conform to the following constraints:

- allocated data can reside in CTR system memory.
- allocated data must be at least 4 bytes aligned.

4.6.3 Number of allocations performed

The `mw::mo::moflex::Demuxer::Create` method does 3 allocations.

Due to the dynamic nature of the `mw::mo::moflex::Demuxer` module, we cannot give a precise number of allocations performed.

The number of allocations done by the `ReadPacket` method are highly variable and depend on the Moflex content and the way the `mw::mo::moflex::Demuxer` module is used by the application.

4.6.4 Amount of memory allocated

Due to the dynamic nature of the `mw::mo::moflex::Demuxer` module, we cannot give a precise maximum amount of memory usage.

The amount of memory allocated for standard scenarios shown in Mobiclip code samples is typically between 100 to 150 KiB, but the amount of memory allocated can vary greatly depending on the following factors:

- the number of demuxed frames you accumulate inside the `mw::mo::moflex::Demuxer` class.
- the kind of video content: for example videos with long static scenes followed by dynamic ones tend to consume significantly more memory.

Reserving 1MiB of memory for the `mw::mo::moflex::Demuxer` module will cover most of your needs with a significant security margin.

5 Mobiclip player programming and multithreading

5.1 Multithreading usage

The Mobiclip SDK API has not been designed to be fully multithread safe: this is to avoid adding critical sections or mutexes overhead in all its functions or methods.

It therefore is the responsibility of the developer to be aware of thread safety issues and to use mutual exclusion mechanisms (like mutexes or critical sections) in its code when necessary.

The Mobiclip SDK API has nevertheless been implemented in ways that allow developers to use it efficiently in a multithreaded environment without asking them to use complex mutual exclusion mechanisms by themselves.

Some Mobiclip SDK modules derived from `mw::mo::GenericVideoDecoder`, like `mw::mo::mobiclip::Decoder`, are therefore implemented in a thread safe way.

5.2 Multithreaded Mobiclip player structure

You will find below a description of the typical structure of a multithreaded Mobiclip player.

Please have a look at the helper `mw::mo::helper::Player` class to for an implementation example of an efficient multithreaded player; they can also be used as is in your own application.

A generic Mobiclip player can be divided into four parts that each have their specific role and that run in their own thread. This brings the number of running threads to four, including the main thread.

These threads can be named as:

- Demux and decode thread.
- Reader thread.
- Video presentation thread.
- Audio presentation thread.

These are described below.

5.2.1 Demux and decode thread.

It does Moflex packet demuxing. It also performs the decoding of audio and video frames; this thread is often the main thread.

The following Mobiclip modules are used:

- `mw::mo::GenericVideoDecoder` for decoding.
- `mw::mo::GenericAudioDecoder` for decoding and giving decoded audio data to Audio presentation thread.
- `mw::mo::moflex::Demuxer`.

5.2.2 Reader thread.

It does Moflex packet reading. This thread is spawned by a specific implementation of a `mw::mo::moflex::Reader` structure like in the `mw::mo::helper::FsReader` class. All reading operations that are not CPU intensive but take time to complete because of slow IO operations (like reading from ROM or MMC/SD/SDHC) are done in this thread.

Read Moflex packets are then given to the `mw::mo::moflex::Demuxer` module of the demux and decode thread.

5.2.3 Video presentation thread.

It retrieves decoded video frames and presents them on screen in a timely manner by using their timestamp value.

The Mobiclip module `mw::mo::GenericVideoDecoder` of the demux and decode thread retrieves video frames data and removes them from the video queue when presentation is done.

Most often this thread is the main thread, that is the thread where developers are doing all their all 3D rendering work.

5.2.4 Audio presentation thread.

It manages the presentation of the decoded audio frames provided by the demux and decode thread.

6 mw::mo::helper namespace API usage guide

The Mobiclip SDK API is giving to the developer a lot of flexibility to perform movie playback for its particular needs but most often what the developer want is only perform a standard movie playback, like playing a 2D or 3D movie on top LCD screen.

For standard scenarios (like the one described before) the use of a higher level API is better suited because it hides a lots of implementation complexity details to the developer and gives him a simpler programming interface to work with.

This section briefly explains how parts of the mw::mo::helper namespace API is implementing a Moflex content playback.

This section does not describe all mw::mo::helper namespace API, only part that helps the developer better understand all processes behind a Moflex content playback.

For a detailed and more complete specification of the parts of the mw::mo::helper API, please see the HTML Function Reference Manual.

6.1 Player state graph

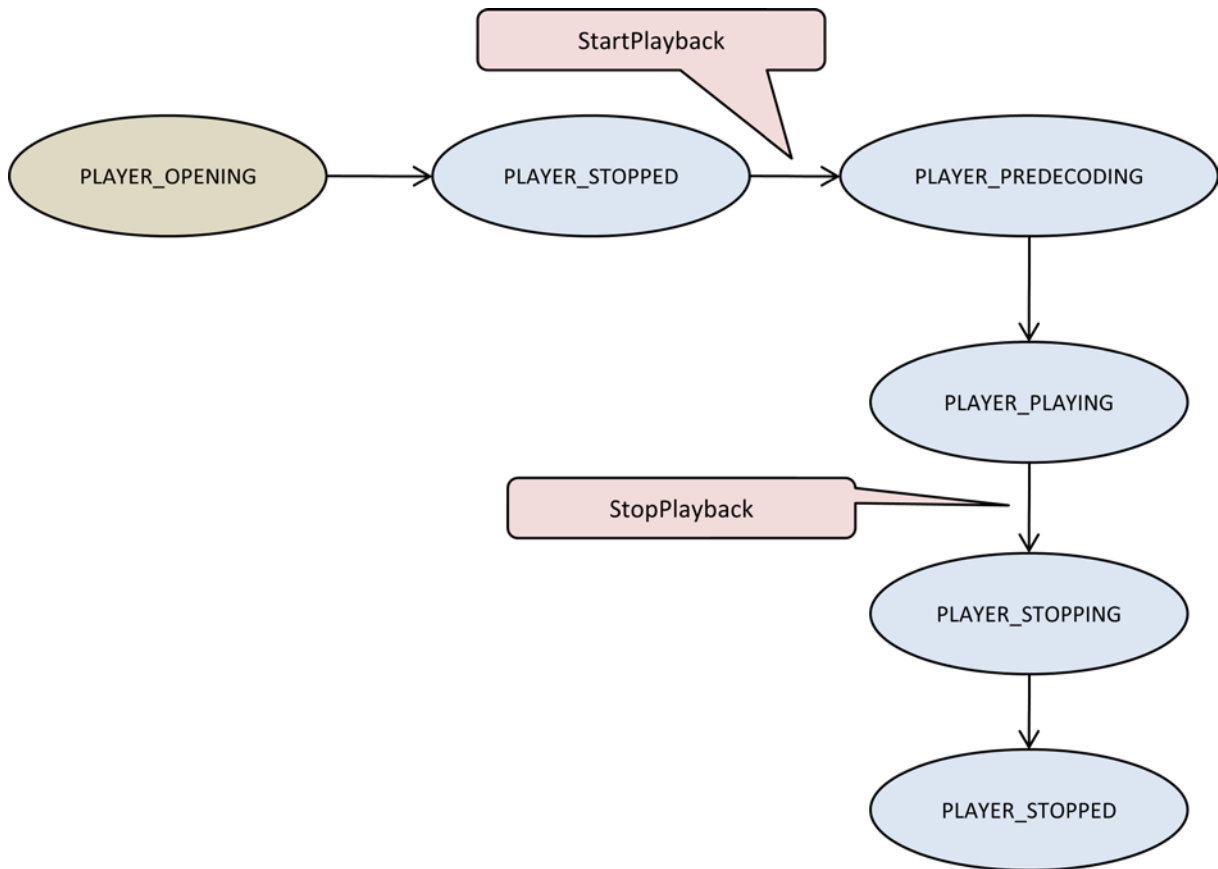
The `mw::mo::helper::Player` class implements a state graph: for a given state and a user action and/or internal event the player knows the next state it will move to.

Player states are defined by `mw::mo::helper::PlayerStatus` enumerated type.

The current player state can be retrieved by calling `mw::mo::helper::Player::GetStatus` method. Please note however that some of the `mw::mo::helper::PlayerStatus` values are never returned by this method, this is because they are intermediate and purely internal player states.

Please find in the figure below a graph showing the states the player goes through when performing a simple movie playback without any user interactions.

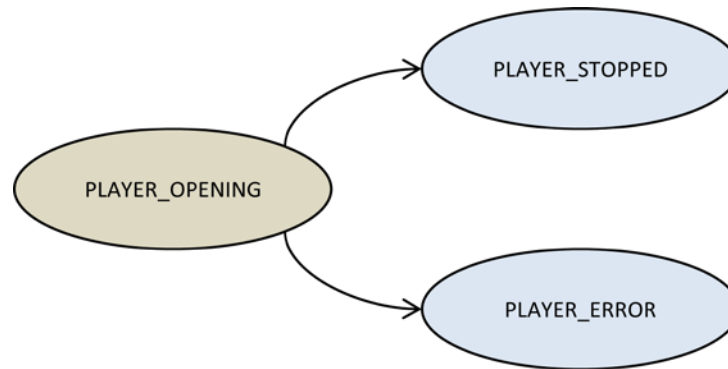
Figure 2 Performing a simple playback



6.1.1 Opening a Moflex content

When a `mw::mo::helper::Player` instance is successfully done using `mw::mo::helper::Player::Create` method, the initial state of the player is `mw::mo::helper::PLAYER_OPENING` (`PLAYER_OPENING` in short) then moves to the `PLAYER_STOPPED` state like in the figure below.

Figure 3 Create call state graph



6.1.2 Starting Moflex content playback

This is done by calling the `mw::mo::helper::Player::StartPlayback` method.

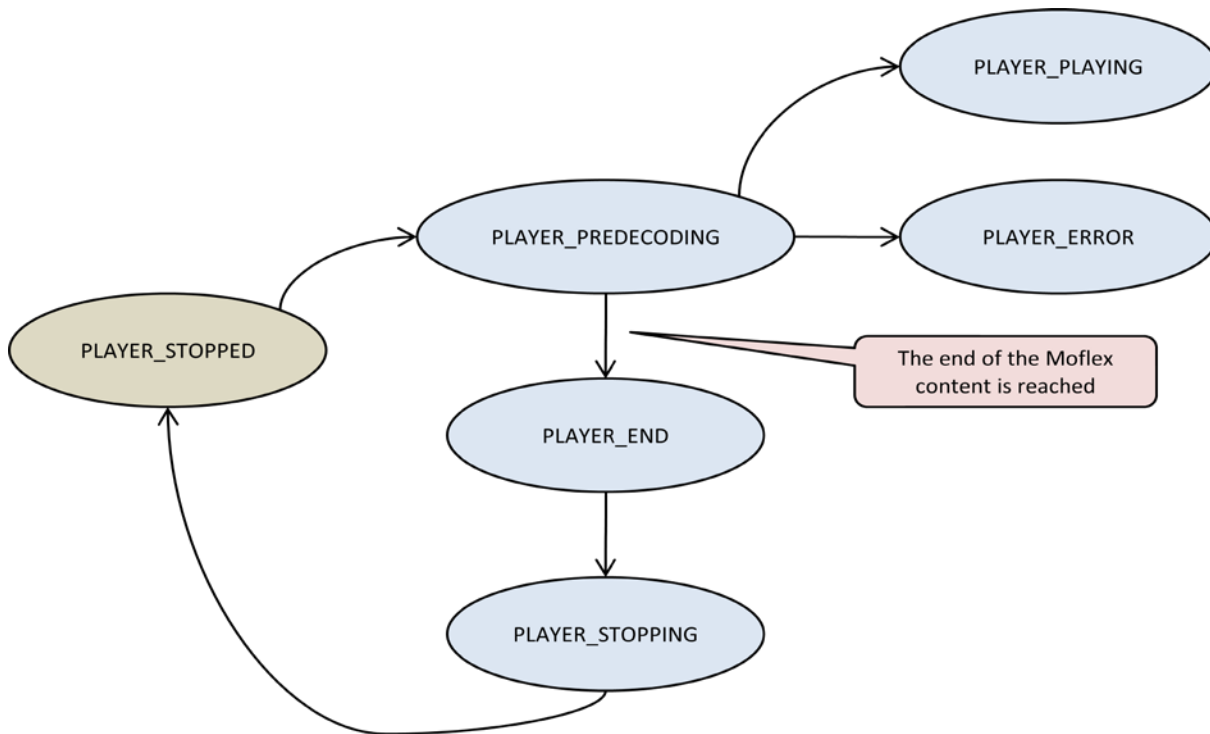
If successful the player then moves to `PLAYER_PREDECODING` state.

While in `PLAYER_PREDECODING` state the player is beginning to decode video and audio frames and enqueue them without launching the playback process.

Once a determined number of decoded video frames are enqueued (it is usually 12 for 3D interleaved videos and 6 otherwise) the playback is started.

This way a good amount of audio data is already buffered and ready to be presented by the audio device, thus greatly limiting the risk of audio underflow problems during playback.

Figure 4 StartPlayback call state graph



6.1.3 Stopping Moflex content playback

This is done by calling the `mw::mo::helper::Player::StopPlayback` method.
If successful the player flushes all enqueued decoded audio and video frames.

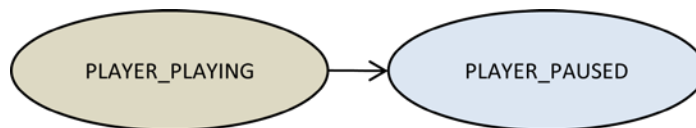
Figure 5 StopPlayback call state graph



6.1.4 Pausing Moflex content playback

This is done by calling the `mw::mo::helper::Player::PausePlayback` method.
Unlike `mw::mo::helper::Player::StopPlayback` method, no enqueued decoded audio and video frames are flushed..

Figure 6 PausePlayback call state graph

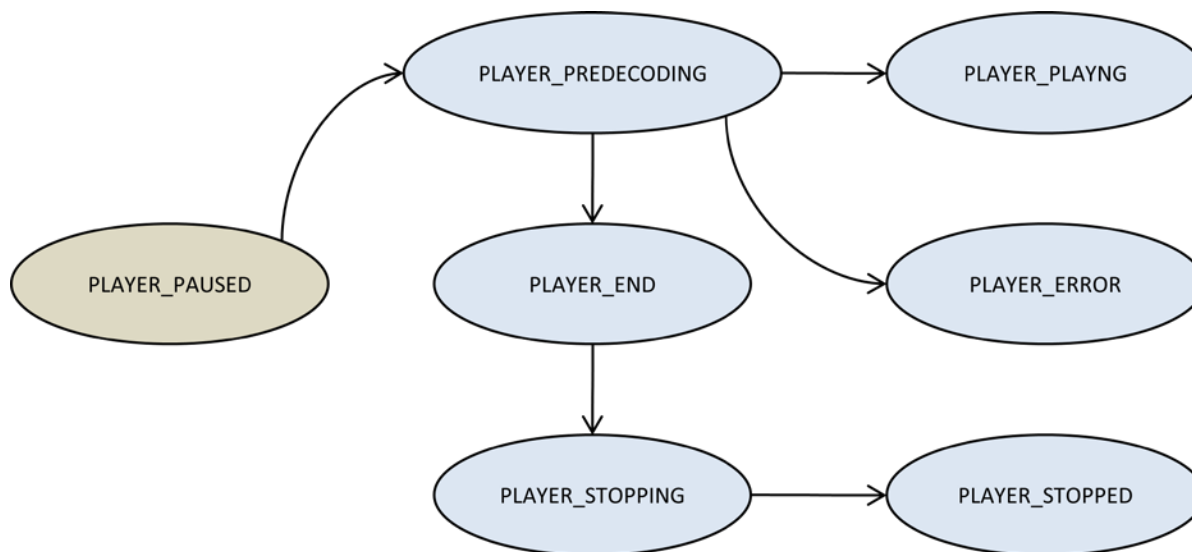


6.1.5 Resuming Moflex content playback

This is done by calling the `mw::mo::helper::Player::ResumePlayback` method.

If successful the player then moves to the `PLAYER_PREDECODING` state and performs the same processes as in the `mw::mo::helper::Player::StartPlayback` method.

Figure 7 ResumePlayback call state graph



6.1.6 Seeking into Moflex content

This is done by calling the `mw::mo::helper::Player::SetPositionByFileOffset` method. Please note that `mw::mo::helper::Player::SetPositionByTimeline` method internally calls it.

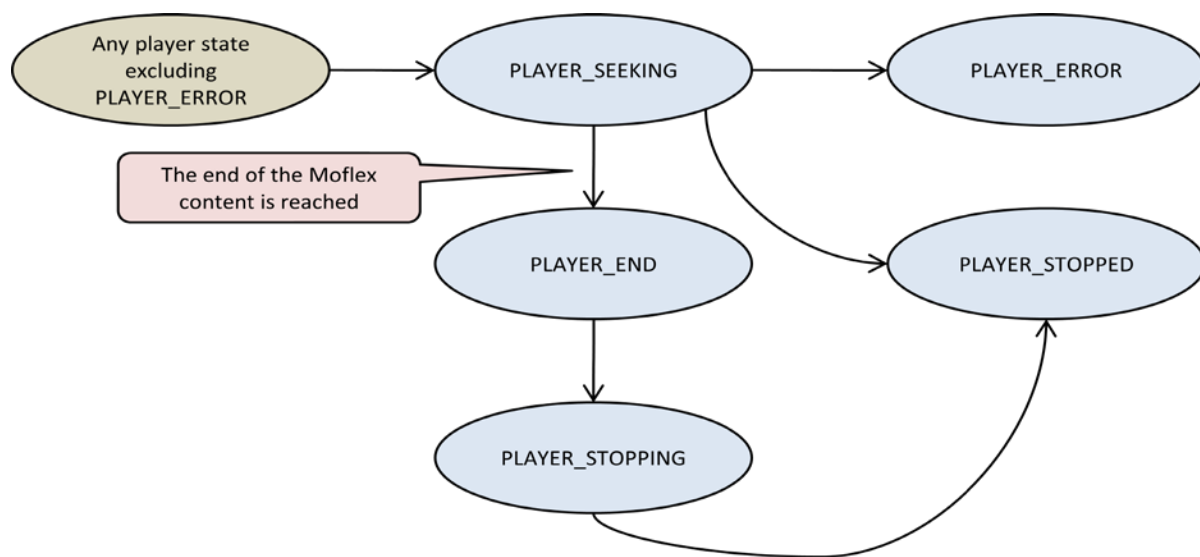
If successful the player moves to the `PLAYER_SEEKING` state.

While in this state it flushes all enqueued decoded audio and video frames.

It will then read new frames from the Moflex content and ignore them until it finds a video frame that is a keyframe.

Once the video keyframe is found frame decoding begins. If the video is stereoscopic interleaved and if the next decoded video frame is not of the same left/right eye pair, the first decoded video frame is removed from the decoder queue without being displayed (this prevents left/right eye inversion problem at display).

Figure 8 SetPositionByFileOffset call state graph



6.2 Performing video presentation

6.2.1 The `mw::mo::helper::VideoPresentationBuffer` class

Video presentation process (i.e. displaying a video image on the 3DS screen) is handled by two components. One of them is a class that is derived from the `mw::mo::helper::VideoPresentationBuffer` class. The buffers managed by this class is then displayed by the game application. These two components works cooperatively to produce display. Examples of this are provided in the sample codes of the SDK.

These `mw::mo::helper::VideoPresentationBuffer` classes give to the `mw::mo::helper::Player` class pointers to RGB image buffers where to perform its YUV to RGB color conversion process. They can manage up to 3 different RGB target buffers (each accessible by a target index), one for each possible display target (that is one for left eye display of top screen, one for right eye display of top screen and one for display of bottom screen).

The RGB information stored in these target buffers are expected to be in 24 bits PICA `GL_RGB_NATIVE_DMP` texture format only (no alpha information, no RGB information encoded into a 16 bit values).

There is one implementation example in the Mobiclip SDK: the `mw::mo::helper::GlVideoPresentation` class show how to perform video presentation using the OpenGL API.

Please note this class is also doing video image rendering into the screen but the developer is free to do that in other parts of its application.

This class is also managing all 3 RGB target buffers but it is not mandatory for a derived `mw::mo::helper::VideoPresentationBuffer` class to do that. For example if the developer wants to display only one 2D video on top or bottom screen it only has to manage 1 RGB target buffer, that simplifies its implementation.

6.2.2 Warning about video presentation performance

The developer must take care of the performance of its video presentation code, that is it must ensure that the display of video frames is not done slower than the video track frame rate.

If not, audio underflow happens during the playback. The problem is that the audio queue is emptied (video is displayed slower than real time but audio is played at the right rate) and the player can't decode new audio frames because its decoded video queue is still full (it is not emptied fast enough because video display is too slow).

Solutions to fix that problem on the application would be:

- When audio underflow is detected, call `mw::mo::helper::Player::StopPlayback` method followed immediately by `mw::mo::helper::Player::StartPlayback` method. This will force the player to flush video queue and perform pre-decoding of new audio and video frames. Doing that will cause an unavoidable momentary stuttering of video display and audio playback.
- Detect when audio buffers are close to be empty (that is before an audio underflow actually happens) then skip the video image presentation. Doing that will cause an unavoidable momentary stuttering of video display.

6.3 Performing audio presentation

6.3.1 The `mw::mo::helper::AudioPresentationBuffer` class

The audio presentation process closely follow the video presentation process. A buffer is provided by a class derived from the `mw::mo::helper::AudioPresentationBuffer` class. However, the player will not directly fill this buffer, instead it will provide data through its API.

This class receives decoded audio frames (in PCM 16 bits format) pushed by the `mw::mo::helper::Player` class, copy them if it has enough place in its playback audio buffer and enqueue them for playback.

There is one implementation example in the Mobiclip SDK: the `mw::mo::helper::PcmAudioPresentation` class show how to perform audio presentation using the `nn::snd` API.

Please note that this class is also doing audio playback but the developer is free to do that in other parts of its application.

6.3.2 Warning about minimum needed audio buffer size

In order for the `mw::mo::helper::Player` class to successfully begin a movie playback, the internal audio buffer of the `mw::mo::helper::AudioPresentationBuffer` class must be big enough to contain all pre-decoded audio frames (audio frames pre-decoding is done at the `mw::mo::helper::Player` `PLAYER_PREDECODING` state). If not the `mw::mo::helper::Player` class will stay in `PLAYER_PREDECODING` state forever and will never move to `PLAYER_STOPPED` state.

Allocating an audio buffer that can contain up to 1 second of audio data (like it is done in the `mw::mo::helper::AudioPresentationBuffer` class) is sufficient for most movie playback scenarios.

Revision History

Version	Revision Date	Category	Description
1.0.7	2012/05/25	-	Note added within 3.6.5 section talking about <code>mw::mo::moflex::Demuxer::ReadPacket</code> method usage.
1.0.6	2012/02/24	-	Added sections 2.4.4 and 4.5 regarding usage of the <code>mw::mo::dspadpcm</code> namespace
1.0.5	2012/01/10	-	- Added Chapter 3.6.8 describing seeking process inside Moflex container using <code>mw::mo::moflex::Demuxer</code> module. - Fixed some memory calculation formulas in Chapter 4. - Updated Chapter 6.2.2 talking about video presentation performance.
1.0.4	2011/10/03	-	- Added chapter 2.2 showing typical Moflex content playback data flow. - Added chapter 2.4.7 talking about <code>mw::mo::helper</code> namespace description. - Added chapter 6 giving a <code>mw::mo::helper</code> namespace API usage guide. - Modification in allocator constraints sections.
1.0.3	2011/07/13	-	Added chapter about Mobiclip SDK module memory usage.
1.0.2	2011/04/08	-	- Added 3.6.1 Moflex lexicon Chapter. - Added sections to describe <code>mw::mo::moflex::StreamInfo</code> enum. - Modifications done in document to match 6.3.1 Chapter. - References to <i>PlayerGeneric</i> and <i>PlayerMobiclip</i> helper classes removed, replaced by <i>Player</i> class.
1.0.1	2010/10/08	-	Changed <i>small</i> and <i>fast</i> strings of library names by (<i>option</i>) string .
1.0.0	2010/08/03	-	Initial version.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

© 2013 Nintendo

The contents of this document cannot be duplicated, copied, reprinted, transferred, distributed, or loaned in whole or in part without the prior approval of Nintendo.